UNIVERSIDAD DE MÁLAGA

Modelado y Diseño de Software

Examen Enero 2024

Nombre: _____

PARTE 1 (15 minutos)

A. Defina los siguientes conceptos [0,-2]
- Ingeniería de software
- Modelo
- Diseño de software
- Patrón de diseño

B. Señala las principales semejanzas y diferencias entre los patrones de diseño "Estado" y "Estrategia" [0.25]

C. Señala las principales semejanzas y diferencias entre los conceptos "estilo arquitectónico" y "patrón de diseño" [0.25]

PARTE 2 (3 horas 45 minutos)

P1. Los coches, sus viajes y sus revisiones

Queremos modelar un sistema que tiene coches de una marca concreta (por ejemplo, TOYOTA) que pueden viajar entre ciudades y se someten a revisiones, de acuerdo con las siguientes especificaciones.

1. El sistema registra ciudades, de modo que a cada par de ciudades las separan unos kilómetros concretos, que son los recorridos que harán los coches de nuestro sistema, y cada ciudad debe estar al menos a 5 kilómetros de distancia de otra.
2. Los coches tienen una fecha de matriculación y registran los kilómetros recorridos. También tienen información sobre si están en garantía y necesitan mantenimiento, como se explica posteriormente.
3. Los coches realizan viajes, de modo que cada viaje tiene una fecha de salida y una fecha de llegada. En un viaje, el coche siempre realiza un recorrido entre dos ciudades.
4. Los coches deben someterse a revisiones, donde toda revisión debe tener una fecha de inicio y una fecha de fin. Una revisión puede tratarse de un trabajo de mantenimiento o bien una reparación de alguna avería. Todas las revisiones deben tener lugar después de que el coche se matriculase.
5. Como mucho, un coche se debe someter a una revisión, como máximo, en un momento dado.
6. Las revisiones se realizan en talleres, que pueden ser oficiales o no. Un taller oficial ofrece una garantía de un tiempo determinado a toda revisión que se realice en su taller. Cada taller se sitúa en una ciudad, y en cada ciudad habrá, a lo sumo, un taller oficial, pudiendo haber varios talleres no oficiales.
7. Si un coche está siendo sometido a una revisión, entonces el coche debe encontrarse en la misma ciudad donde está el taller.
8. En relación con los viajes que realiza un coche, todo viaje terminado tendrá una fecha de salida y una fecha de llegada. Si el coche está realizando actualmente algún viaje, dicho viaje únicamente tendrá fecha de salida, pero no de llegada.
9. Un coche se encontrará en todo momento bien realizando un viaje determinado o bien en una ciudad.
10. Si el coche ha completado al menos un viaje y no se encuentra viajando, entonces debe encontrarse en la ciudad a la que llegó en su último viaje.
11. Dos viajes no pueden solaparse en el tiempo, es decir, un viaje debe ocurrir siempre después de otro, pudiendo la fecha de llegada de un viaje coincidir con la fecha de salida del siguiente.
12. Las ciudades de origen y destino de los viajes deben ser coherentes. Es decir, si un coche realiza un viaje desde la ciudad A hasta la ciudad B, el próximo viaje debe partir desde la ciudad B.
13. Los kilómetros que tiene un coche deben ser la suma de los kilómetros de los viajes que dicho coche ha completado. En nuestro sistema, no nos debemos preocupar de insertar manualmente el número de kilómetros de un coche, pues debería estar actualizado en todo momento.
14. En este sistema, todas las fechas serán un valor entero que representa un día. Vamos a suponer que tanto los viajes como las revisiones comienzan un día y terminan X días posteriores. Asimismo, por simplicidad, vamos a suponer que un año son 100 días.
15. En el punto 2 se mencionó que los coches tienen información sobre si están en garantía. Un coche está en garantía si no han pasado 4 años desde que se matriculó. También está en garantía si han pasado menos días que los indicados en la garantía del taller oficial donde el coche tuvo alguna revisión.
16. En el punto 2 se mencionó que los coches tienen información sobre si necesitan mantenimiento. Durante los primeros cuatro años desde la fecha de matriculación, un coche no necesita mantenimiento. Después de ese tiempo, un coche no

necesita mantenimiento hasta pasado un año de la última revisión de mantenimiento que tuviera, sin importar el taller donde se realizó.

Se pide:

(a) Desarrollar en UML, utilizando la herramienta USE, un modelo conceptual de la estructura de dicho sistema, con los correspondientes elementos, las relaciones entre ellos y todas las restricciones de integridad necesarias. En lo relacionado con el tiempo, en este apartado podemos considerar que tenemos una clase *Clock* de la cual habrá una única instancia. Dicha clase tiene un atributo "now" que nos indica el día en que nos encontramos actualmente. No hay que modelar el paso del tiempo en este apartado. ***Se debe entregar la imagen del diagrama de clases y todo el código USE desarrollado. Si durante el desarrollo se ha creado un modelo conceptual en SOIL para comprobar el funcionamiento del sistema, se puede entregar también el código SOIL correspondiente (y cualquier imagen del modelo conceptual necesaria para explicarlo).***

(b) Este apartado consiste en modelar el comportamiento del sistema. En particular, se va a modelar únicamente el comportamiento de los coches cuando viajan. Para ello, vamos a registrar la *velocidad* de los coches. La velocidad de un coche indica cuántos kilómetros avanza cada día en los viajes. El sistema también va a almacenar los *kilómetros* de los viajes. Cuando un viaje está en curso, este atributo indica los kilómetros que lleva realizado. Cuando un viaje se completa, este atributo debe tener el mismo valor que los kilómetros del recorrido que ha realizado. Se deben modelar las siguientes acciones:

(b1) Un coche comienza un viaje desde la ciudad en la que se encuentra. Esta operación debe recibir como parámetro el recorrido entre dos ciudades que debe realizar en su viaje.

(b2) Una operación avanzar que se ejecuta sobre los coches, y que no recibe ningún parámetro. Esta operación debe hacer avanzar el coche el número de kilómetros indicados en su velocidad si el coche está realizando algún viaje.

(b3) Se debe modelar el paso del tiempo, de modo que un tick del reloj representa el paso de un día, lo cual se debe tener en cuenta a la hora de que los coches puedan avanzar en el viaje que estén realizando.

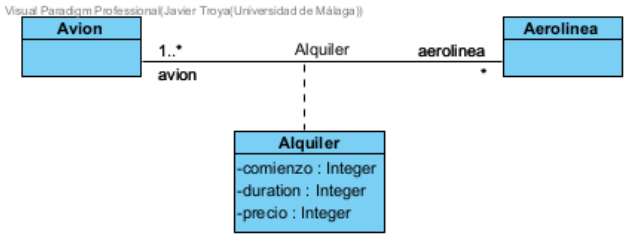
Se debe entregar la imagen del diagrama de clases y el código USE desarrollado (al ser este apartado un incremento respecto del anterior, hay que entregar únicamente el código USE nuevo). En las operaciones añadidas, especificar las pre- y post- condiciones.

(c) Ahora se debe desarrollar un modelo de objetos y simularlo. Vamos a considerar tres ciudades: Málaga, Sevilla y Granada. Tendremos dos recorridos: entre Málaga y Sevilla con 210 kilómetros, y entre Sevilla y Granada con 250 kilómetros. Supondremos un coche matriculado en el instante 0 (día 0) y que viaja a una velocidad de 27. El coche comienza en Málaga y continúa en Málaga hasta el día 5, día en que comienza un viaje haciendo el recorrido de Málaga a Sevilla. Los días van pasando y el coche va avanzando hasta que llega a Sevilla. El mismo día que llega a Sevilla, el coche comienza otro viaje haciendo el recorrido entre Sevilla y Granada. Los días van pasando y el coche continúa realizando el viaje. Una vez llega a Granada, la simulación termina.

Se debe mostrar 3 imágenes del diagrama conceptual: una en el instante 0, otra cuando el coche llega a Sevilla y otra cuando el coche lleva a Granada. Entregar también el código SOIL necesario para reproducir el modelo conceptual y la simulación.

P2. Los aviones y las aerolíneas

Tenemos un sistema donde las aerolíneas no tienen aviones en posesión, sino que los alquilan con una duración determinada, expresada en años. El siguiente modelo conceptual representa este sistema:



Los atributos de la clase *Alquiler* indican el año de comienzo, la duración expresada en años y el precio mensual de cada alquiler. Deben cumplirse los siguientes requisitos:

1. Los atributos "precio", "duracion" y "comienzo" de la clase *Alquiler* deben ser estrictamente positivos.
2. Un avión no puede tener más de dos alquileres que superen los 4 años, aunque puede tener otros muchos de duración menor.

3. Una aerolínea no se puede permitir tener más del 20% de los alquileres sus aviones con una duración superior a 5 años.
4. Los años de comienzo de los alquileres de un avión no pueden coincidir, pues se supone que la adaptación de los aviones a las distintas aerolíneas debe ser paulatina.

Se pide:

(a) Convertir el modelo conceptual en un modelo de diseño, utilizando la herramienta Visual Paradigm, que permita la implementación, en un lenguaje orientado a objetos como Java, de las entidades, asociaciones y restricciones correspondientes. Además de las restricciones impuestas por el modelo conceptual, deben incluirse en el diagrama las restricciones OCL expresadas anteriormente.

(b) Desarrolle una implementación en Java correspondiente a dicho modelo de diseño, que incorpore el código de andamiaje para crear objetos y relaciones entre ellos garantizando en todo momento la coherencia de los objetos y las relaciones de la implementación, incluidas las restricciones.

- **Puntuaciones**: Parte 1: 0.5; Parte 2: P1 (4+2+0.5) P2 (1+2)
- **Examen reducido**: Problemas P1(a) y P2(b) – la nota obtenida se pondera a 9.5
- **Entrega**: La entrega se hará a través del campus virtual, en **dos archivos.** Por un lado, se entregará, en **formato PDF**, un archivo que contendrá una memoria explicativa del examen y en donde se describirán las soluciones propuestas para cada problema y se incluirán todas las imágenes con los diagramas UML, así como todos los códigos en Java y USE desarrollados. El otro archivo será un **comprimido ZIP** donde se incluyan todos los proyectos USE, Java y Visual Paradigm desarrollados.

SOLUCIONES

A. Defina los siguientes conceptos [0,-2]

- La *ingeniería de software* es la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento de software.
- *Modelo*: Una representación o especificación de un sistema, desde un determinado punto de vista y con un objetivo concreto.
- *Diseño*: Conjunto de planes y decisiones para definir un producto con los suficientes detalles como para permitir su realización física de acuerdo a unos requisitos
- *Patrón de Diseño*: Una solución probada que se puede aplicar con éxito a un determinado tipo de problemas que aparecen repetidamente en el desarrollo de software.

B. Señala las principales semejanzas y diferencias entre los patrones de diseño "Estado" y "Estrategia" [0.25]

El patrón Estrategia se utiliza cuando se dispone de diferentes algoritmos (o de diferentes versiones de un mismo algoritmo) para una tarea específica y el cliente decide la implementación concreta que se utilizará en tiempo de ejecución. De esta forma es posible cambiar la implementación de un método en tiempo de ejecución, dependiendo de diferentes criterios (tanto internos del propio objeto como externos a él, o del propio diseño de la aplicación). Este patrón permite variar los algoritmos haciéndolos independientes de los clientes que los utilizan.

El patrón Estado se utiliza cuando el comportamiento del objeto depende de su estado actual. Para cada posible estado se define el comportamiento del objeto, que cambia cuando este cambia de estado.

Como diferencias, el patrón Estrategia se usa a nivel de método (definiendo varias implementaciones de un mismo método, mientras que el Estado es a nivel de objeto (definiendo la implementación de las operaciones de un objeto en un estado dado del propio objeto)

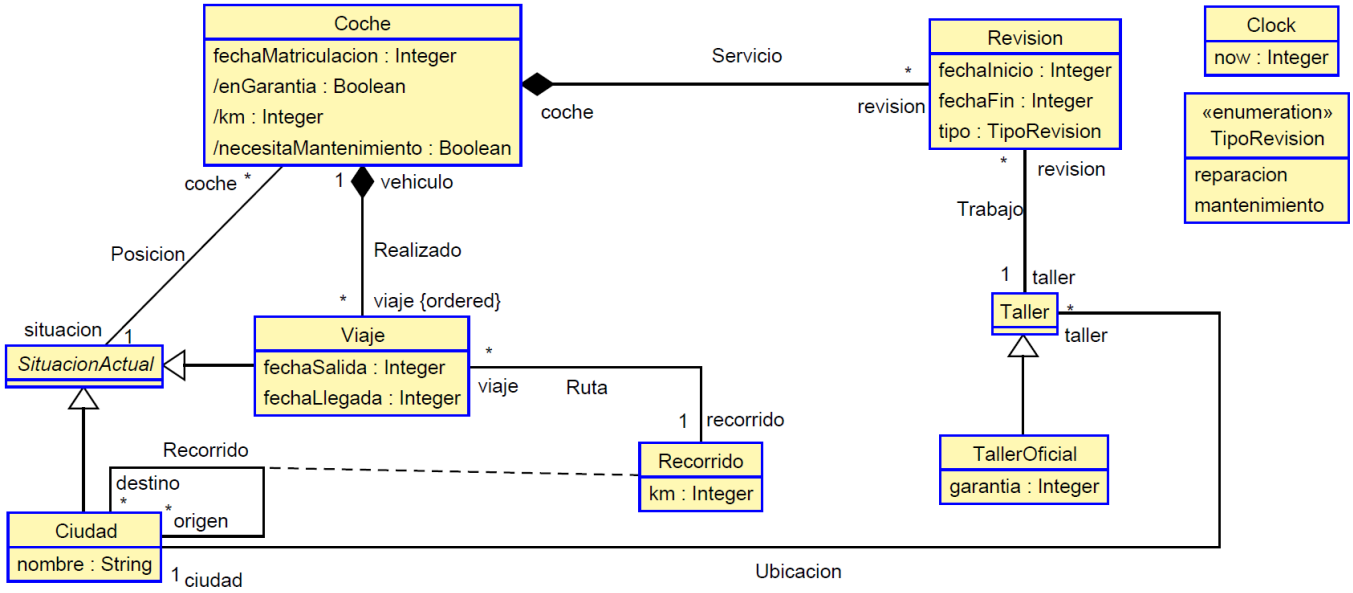
Por supuesto, ambos patrones pueden combinarse, por ejemplo, haciendo que dentro de un mismo estado se puedan usar distintas estrategias de implementación de un método, dependiendo de algunos criterios

C. Señala las principales semejanzas y diferencias entre los conceptos "estilo arquitectónico" y "patrón de diseño" [0.25]

[Sobre todo lo del nivel de abstracción…]

P1. Los coches

Apartado (a) Un posible modelo del sistema pedido viene representado por el siguiente diagrama de clases en UML



El correspondiente código USE, que también contiene las restricciones de integridad expresadas en el enunciado, es el siguiente:

```
model Coches

enum TipoRevision {reparacion,mantenimiento}

class Clock
attributes
    now:Integer init:0
end

--abstract class Taller
--end

class Taller
end

class TallerOficial < Taller
attributes
    garantia : Integer
end

class Revision
attributes
    fechaInicio : Integer
    fechaFin : Integer
    tipo : TipoRevision
end

class Coche
attributes
    fechaMatriculacion : Integer
```

```
    enGarantia : Boolean derive : let ahora : Integer = Clock.allInstances()->asSequence()->first().now in
                    ahora - self.fechaMatriculacion <= 400
            or self.revision->select(r|r.taller.oclIsTypeOf(TallerOficial))->exists(r|ahora-
r.fechaFin<=r.taller.oclAsType(TallerOficial).garantia)
    km : Integer derive : self.viaje->select(v|not v.fechaLlegada.oclIsUndefined)->collect(v|v.recorrido.km)->sum()
    averiado : Boolean
    necesitaMantenimiento : Boolean derive : let ahora : Integer = Clock.allInstances()->asSequence()->first().now
in
                    ahora - self.fechaMatriculacion > 400
                    and not self.revision->exists(r|r.tipo=#mantenimiento and ahora-r.fechaFin<=100)
end

abstract class SituacionActual
end

class Ciudad < SituacionActual
attributes
    nombre : String
end

class Viaje < SituacionActual
attributes
    fechaSalida : Integer
    fechaLlegada : Integer
end

associationclass Recorrido between
    Ciudad[*] role origen
    Ciudad[*] role destino
    attributes
        km : Integer
end


composition Servicio between
    Coche [1] role coche
    Revision [*] role revision
end

association Trabajo between
    Revision [*] role revision
    Taller [1] role taller
end

association Posicion between
    Coche[*] role coche
    SituacionActual[1] role situacion
end

association Ruta between
    Viaje[*] role viaje
    Recorrido[1] role recorrido
```

```
end

composition Realizado between
    Coche[1] role vehiculo
    Viaje[*] role viaje ordered
end

association Ubicacion between
    Taller[*] role taller
    Ciudad[1] role ciudad
end



----------Invariantes------------
constraints

context Coche
  inv KmRealizados : --Requisito 13. Los km del coche deben ser iguales a la suma de los viajes realizados
      self.km = self.viaje->select(v|not v.fechaLlegada.oclIsUndefined)->collect(v|v.recorrido.km)->sum()
  inv SituacionActualCorrecta : --Requisitos 8, 9 y 10. Si el coche está realizando un viaje, debe encontrarse en ese viaje, sino, está en la última ciudad a la que ha viajado. Si no ha realizado viajes aún, se encuentra en cualquier ciudad
      if self.viaje->size()=0 --Si no ha hecho ningún viaje, entonces se encuentra en cualquier ciudad
      then self.situacion.oclIsTypeOf(Ciudad)
      else if self.viaje->forAll(v|not v.fechaLlegada.oclIsUndefined) -- Si ha completado todos los viajes, se tiene que encontrar en una ciudad
           then self.situacion = self.viaje->select(v1|self.viaje->forAll(v2|v1.fechaLlegada>=v2.fechaLlegada))->asSequence()->first().recorrido.destino
           else self.situacion = self.viaje->select(v|v.fechaLlegada.oclIsUndefined)->asSequence()->first()
           endif
      endif
  inv ViajesCompletados : --Requisito 8. Todos los viajes deben tener fecha de salida y llegada salvo, a lo sumo, uno
      self.viaje->select(v|v.fechaLlegada.oclIsUndefined)->size()<=1
  inv RevisionesDespuesMatriculacion : --Requisito 4
      self.revision->forAll(r | r.fechaInicio > self.fechaMatriculacion)
  inv ViajesNoSolapados : --Requisito 11. Dos viajes no pueden solaparse en el tiempo
      self.viaje->forAll(v1,v2 | v1<>v2 implies v1.fechaSalida<>v2.fechaSalida and
                                                (v2.fechaSalida>=v1.fechaLlegada or
v1.fechaSalida>=v2.fechaLlegada))
  inv ViajesEnSecuencia : --Requisito 12. En los viajes de cada coche, la ciudad de origen de un viaje debe ser la ciudad destino del último viaje realizado anterior a éste
      Sequence{1..self.viaje->size-1}->forAll(i|self.viaje->at(i).recorrido.destino=self.viaje->at(i+1).recorrido.origen)
  inv MaximoUnaRevisionActualmente : -- Requisito 5. El coche debe estar efectuando como mucho una revisión
      self.revision->select(r|r.fechaFin=null)->size()<=1
  inv CiudadRevision : --Requisito 7. Si está en una revisión, el coche debe encontrarse en la ciudad donde esté el taller
      if (self.revision->select(r|r.fechaFin=null)->size()=1)
      then self.situacion.oclAsType(Ciudad) = self.revision->select(r|r.fechaFin=null)->asSequence()->first().taller.ciudad
```

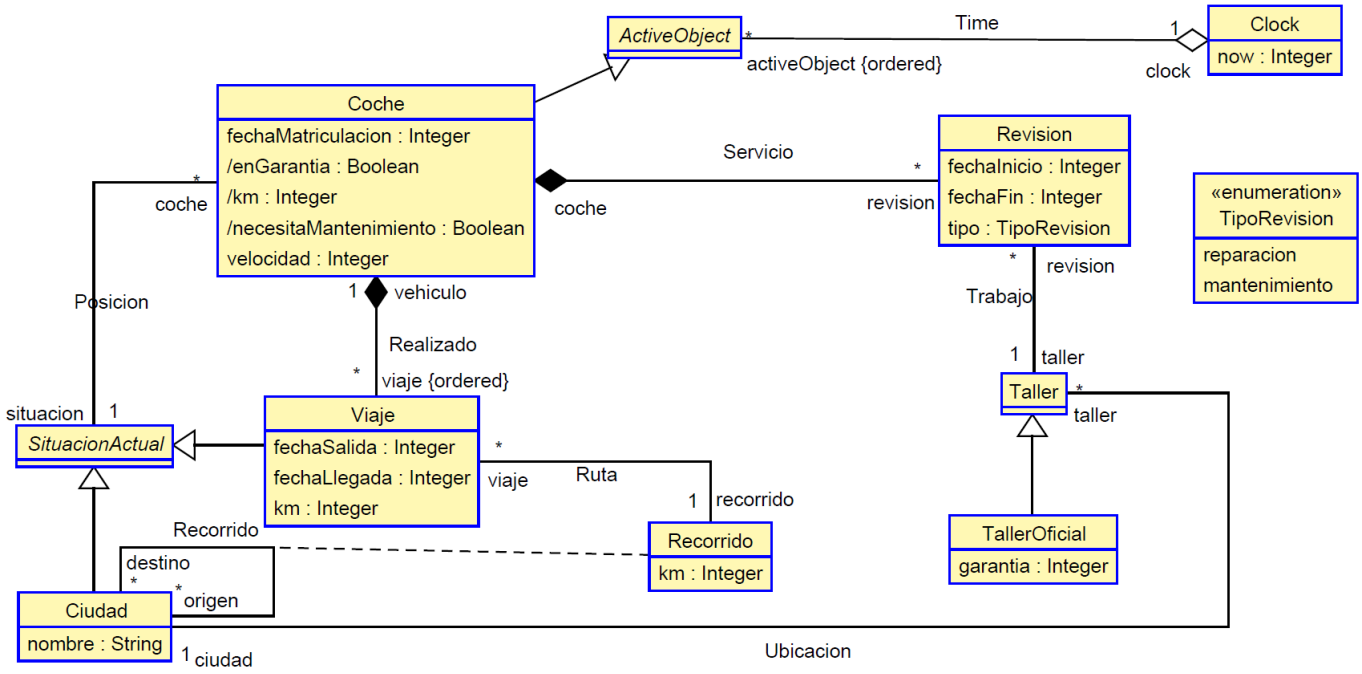
```
      else true
      endif


context Recorrido
  inv CiudadesDiferentes : -- Requisito 1. Las ciudades de origen y destino de un Recorrido deben ser diferentes
      self.origen <> self.destino
  inv SeparacionEntreCiudades : -- #Requisito 1. Toda ciudad está separada al menos 5 km
      self.km >= 5

context Ciudad
  inv UnicoTallerOficial : -- Requisito 6. En cada ciudad hay como mucho un solo taller oficial
      self.taller->select(t|t.oclIsTypeOf(TallerOficial))->size()<=1
```

A modo de ejemplo, el siguiente fichero SOIL describe una serie de acciones para crear un modelo del sistema:

```
reset

!new Clock('clock')
!clock.now := 500

!new Ciudad('Malaga')
!new Ciudad('Sevilla')
!new Ciudad('Granada')

!insert(Malaga,Sevilla) into Recorrido --Recorrido1
!Recorrido1.km := 200
!insert(Sevilla,Granada) into Recorrido --Recorrido2
!Recorrido2.km := 240
!insert(Granada,Malaga) into Recorrido --Recorrido3
!Recorrido3.km := 120
--!insert(Granada,Granada) into Recorrido -- Para invariante "CiudadesDiferentes"
--!Recorrido4.km := 0

!new Viaje('v1')
!v1.fechaSalida := 1
!v1.fechaLlegada := 2
!insert(v1,Recorrido1) into Ruta

!new Viaje('v2')
!v2.fechaSalida := 2
!v2.fechaLlegada := 4
!insert(v2,Recorrido2) into Ruta

!new Viaje('v3')
!v3.fechaSalida := 5
--!v3.fechaLlegada := 7
!insert(v3,Recorrido3) into Ruta

!new Coche('c1')
!c1.fechaMatriculacion := 10
```

```
!insert(c1,v1) into Realizado
!insert(c1,v2) into Realizado
!insert(c1,v3) into Realizado

--!insert(c1,Sevilla) into Posicion
!insert(c1,v3) into Posicion

!new Coche('c2')
!c2.fechaMatriculacion := 310

!insert(c2,Malaga) into Posicion

!new Taller('t1')
!new TallerOficial('to1')
!to1.garantia:=100

!insert(t1,Malaga) into Ubicacion
!insert(to1,Sevilla) into Ubicacion

!new Revision('r1')
!r1.fechaInicio:=410
!r1.fechaFin:=412
!r1.tipo:=#mantenimiento

!insert(r1,t1) into Trabajo
!insert(c1,r1) into Servicio

!new Coche('c3')
!c3.fechaMatriculacion := 20
!insert(c3,Malaga) into Posicion
!new Revision('r2')
!r2.fechaInicio:=495
!r2.tipo:=#mantenimiento
!insert(r2,t1) into Trabajo
!insert(c3,r2) into Servicio
```

Apartado (b) Un posible modelo del sistema pedido viene representado por el siguiente diagrama de clases en UML



Se ha añadido lo necesario para modelar el tiempo:

```
class Clock
attributes
    now:Integer init:0
operations
    tick()
    begin
        self.now:=self.now+1;
        for o in self.activeObject do
            o.tick();
        end
    end
    post TimePasses: now = now@pre+1
    run(n:Integer)
    begin
        for i in Sequence{1..n} do
            self.tick()
        end
    end
    pre NoTimeTravels: n>0
    post TimeFlies: now = now@pre+n
end -- class Clock

abstract class ActiveObject
operations
    tick()
    begin
    end
end

aggregation Time between
Clock [1] role clock
```

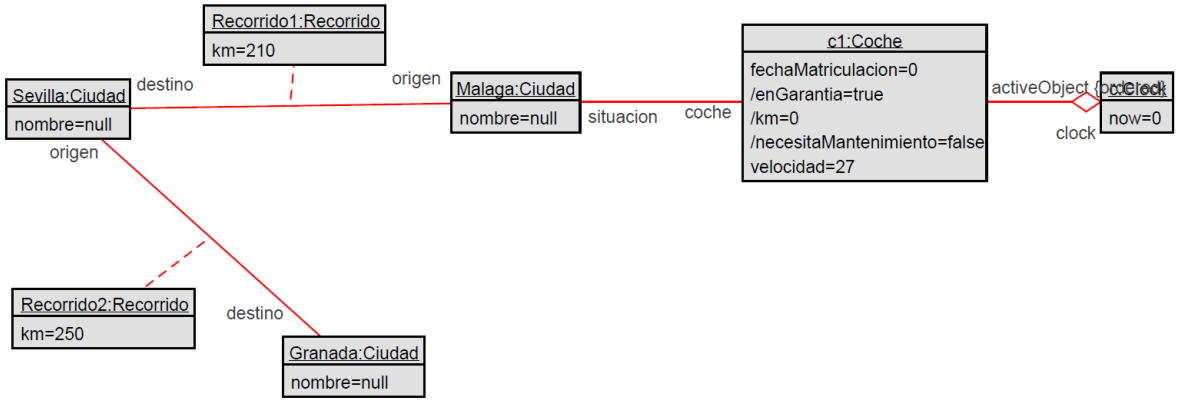
```
ActiveObject [*] role activeObject ordered
end
```

Además, aparte de los atributos *velocidad* en la clase Coche y *km* en la clase Viaje, se han añadido las siguientes operaciones sobre la clase Coche (clase que ahora hereda de ActiveObject):

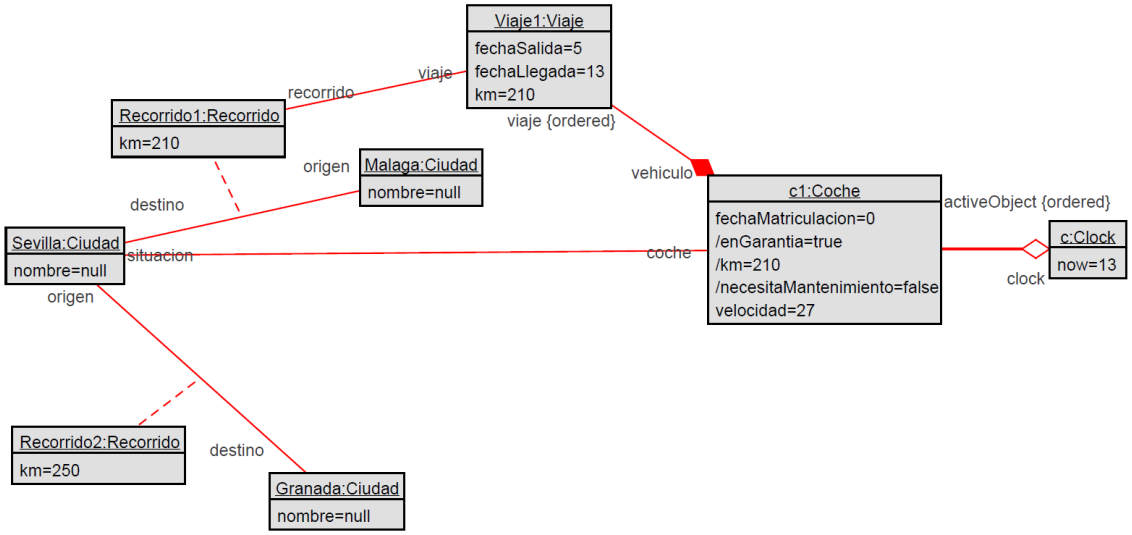
```
operations
    iniciarViaje(r:Recorrido)
    begin
        declare v : Viaje;
        v := new Viaje();
        v.fechaSalida := Clock.allInstances()->asSequence()->first().now;
        insert(v,r) into Ruta;
        insert(self,v) into Realizado;
        delete(self,r.origen) from Posicion; --Ya no está en la ciudad
        insert(self,v) into Posicion; --Se encuentra de viaje
    end
    pre recorridoExiste: (r <> null and r.origen<>null and r.destino<>null)
    pre seEcuentraEnOrigen: (self.situacion.oclAsType(Ciudad)=r.origen)
    post unViajeMas: (self.viaje->size() = self.viaje@pre->size()+1)
    avanzar()
    begin
        declare v : Viaje, c : Clock;
        c := Clock.allInstances()->asSequence()->first();
        v := self.viaje->select(a|a.fechaLlegada=null)->first();
        if (v.km + self.velocidad) < v.recorrido.km
        then v.km := v.km + self.velocidad --Sigue viajando
        else v.km:=v.recorrido.km; --Ha llegado
             v.fechaLlegada:=c.now; --Es el instante al que transita
             delete(self,v) from Posicion; --Ya no está viajando
             insert(self,v.recorrido.destino) into Posicion --Ha llegado a la ciudad
        end
    end
    pre estaViajando: (self.viaje->exists(v|v.fechaLlegada=null))
    post haAvanzado: (self.viaje->collect(km)->sum() > self.viaje@pre->collect(km@pre)->sum())
    tick()
    begin
        if self.viaje->exists(v|v.fechaLlegada=null)
        then self.avanzar();
        end
    end
```

Apartado c). Las imágenes de los tres modelos conceptuales son:

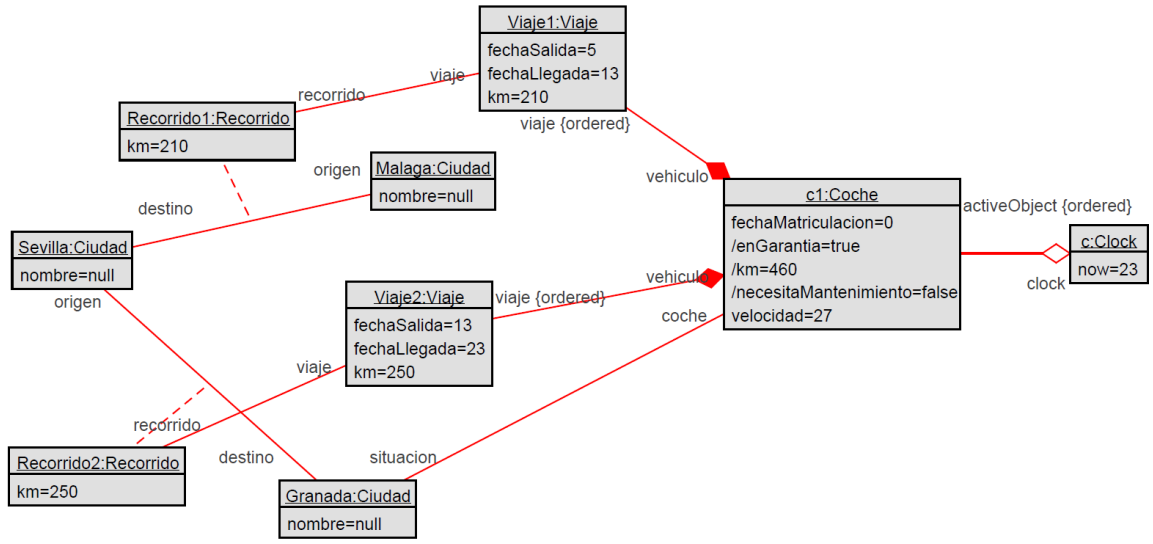
Instante 0:



Instante 13: el coche ha llegado a Sevilla:



Instante 23: el coche ha llegado a Granada:



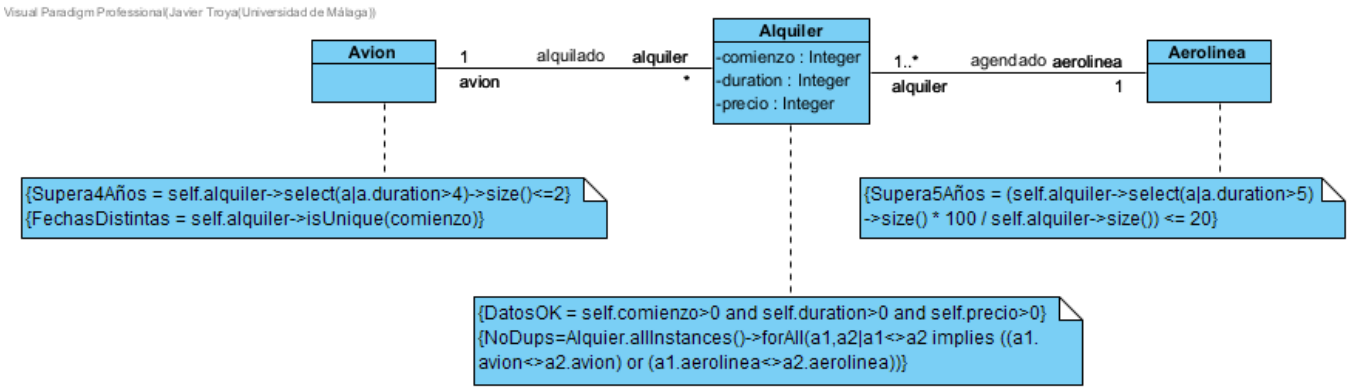
El código SOIL necesario para reproducir esta simulación es:

```
reset

!new Ciudad('Malaga')
!new Ciudad('Sevilla')
!new Ciudad('Granada')

!insert(Malaga,Sevilla) into Recorrido --Recorrido1
```

```
!Recorrido1.km := 210
!insert(Sevilla,Granada) into Recorrido --Recorrido2
!Recorrido2.km := 250

!new Coche('c1')
!c1.fechaMatriculacion := 0
!c1.velocidad := 27
!insert(c1,Malaga) into Posicion

!new Clock('c')
!insert(c,c1) into Time

-----------------------SISTEMA INICIALIZADO-----------------
-- Lo de abajo son los comandos que hay que ejecutar
!c.run(5)
!c1.iniciarViaje(Recorrido1)
!c.tick()
!c.tick()
!c.tick()
!c.tick()
!c.tick()
!c.tick()
!c.tick()
!c.tick()
!c1.iniciarViaje(Recorrido2)
!c.tick()
!c.tick()
!c.tick()
!c.tick()
!c.tick()
!c.tick()
!c.tick()
!c.tick()
!c.tick()
!c.tick()
```

P2. Los aviones y las aerolíneas

a) Modelo de diseño con las restricciones:



b) La implementación en Java correspondiente al modelo de diseño anterior es la siguiente.

```
//------------------
// CLASS Avion
//------------------

package aviones;

import empresas.ContratoEnCurso;
import empresas.Empresa;

import java.util.ArrayList;
import java.util.Enumeration;
import java.util.List;

public class Avion {
    private List<Alquiler> alquiler;

    public Avion(){
        alquiler = new ArrayList<Alquiler>();
    }
    protected void addAlquiler(Alquiler a){alquiler.add(a);}
    protected void rmAlquiler(Alquiler a){alquiler.remove(a);}
    public Enumeration<Alquiler> getAlquiler(){
        return (Enumeration<Alquiler>) alquiler;
    }
    protected int alquilerMayor4Años(){
        Enumeration<Alquiler> it = this.getAlquiler();
        int cont = 0;
        while(it.hasMoreElements()){
            if (it.nextElement().duration>4) cont++;
        }
        return cont;
    }

    protected boolean alquiladoEn(Aerolinea a){
        Enumeration<Alquiler> it = this.getAlquiler();
        while(it.hasMoreElements()){
            if (it.nextElement().aerolinea==a) return true;
        }
        return false;
    }

    protected boolean sinSolapamientos(){
        Enumeration<Alquiler> it1 = this.getAlquiler();
        Enumeration<Alquiler> it2 = this.getAlquiler();
        while (it1.hasMoreElements()){
            while (it2.hasMoreElements()){
                if (it1.nextElement()!=it2.nextElement() &&
                        it1.nextElement().comienzo==it2.nextElement().comienzo)
                    return false;
            }
        }
        return true;
    }

}


//------------------
// CLASS Aerolinea
//------------------

package aviones;

import java.util.ArrayList;
import java.util.List;
import java.util.Enumeration;

public class Aerolinea {
    private List<Alquiler> alquiler;

    public Aerolinea(Avion a, int precio, int duracion, int comienzo){
```

```
        //necesita al menos un avión y por tanto se crea el alquiler
        //la duración debe ser menor de 5 años para garantizar que menos del 20% de alquileres
son
        //por más de 5 años
        assert(duracion < 5);
        alquiler = new ArrayList<Alquiler>();
        Alquiler al = new Alquiler(a, this, precio, comienzo, duracion);
        //no hace falta añadir al a la lista, se hace directaemente al crear el alquiler
    }
    protected void addAlquiler(Alquiler a){alquiler.add(a);}
    protected void rmAlquiler(Alquiler a){alquiler.remove(a);}
    public Enumeration<Alquiler> getAlquiler(){
        return (Enumeration<Alquiler>)alquiler;
    }

    //Método adicional para comprobar la restricción 3: menos del 20% de alquileres con
duración > 5 años
    protected boolean checkDuration(){
        Enumeration<Alquiler> it = this.getAlquiler();
        int count=0, total=0;
        while (it.hasMoreElements()){
            total++;
            if (it.nextElement().duration>5) count++;
        }
        return(count*100/total)>20;
    }
}

//------------------
// CLASS Alquiler
//------------------

package aviones;

public class Alquiler {
    int precio;
    int duration;
    int comienzo;
    Avion avion;
    Aerolinea aerolinea;
    public Alquiler(Avion a, Aerolinea ae, int precio, int comienzo, int duration){
        assert (precio>0); //restricción 1
        assert (comienzo>0); //restricción 1
        assert (duration>0); //restricción 1
        assert (duration<=4 || a.alquilerMayor4Años()<2); /*restricción (2) : o bien el nuevo
alquiler que
        se está creando tiene una duración menor a 4 años, o bien el Avión para el que se está
creando el alquier no
        tiene ya dos alquileres por menos de 2 años. Así nos aseguramos que, como mucho, éste
tenga 2 alquileres
        con duración mayor a 4 años.*/
        assert (!avion.alquiladoEn(ae)); //duplicidad de alquileres

        this.duration = duration;
        this.comienzo = comienzo;
        this.precio = precio;
        this.avion = a;
        this.aerolinea = ae;
        a.addAlquiler(this);
        ae.addAlquiler(this);

        assert(a.sinSolapamientos()); //Restricción 4
        assert (ae.checkDuration()); //Restricción 3
    }

    public int getPrecio() {
        return this.precio;
    }

    public int getDuration(){
        return this.duration;
    }

    public int getComienzo(){
```

```
        return this.comienzo;
    }
}
```